NASA/CR-2001-210873
ICASE Report No. 2001-15

A Component-based Programming Model for Composite, Distributed Applications

Thomas M. Eidson
ICASE, Hampton, Virginia

ICASE
NASA Langley Research Center
Hampton, Virginia

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

May 2001

Form SF298 Citation Data

Report Date *("DD MON YYYY")* 00MAY2001	**Report Type** N/A	**Dates Covered (from... to)** *("DD MON YYYY")*

Title and Subtitle A Component-based Programming Model for Composite, Distributed Applications	**Contract or Grant Number**
	Program Element Number
Authors Thomas M. Eidson	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) NASA Langley Research Center Hampton, Virginia 26381-2199	**Performing Organization Number(s)**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Monitoring Agency Acronym**
	Monitoring Agency Report Number(s)

Distribution/Availability Statement
Approved for public release, distribution unlimited

Supplementary Notes
ICASE Report No. 2001-15

Abstract
Abstract. The nature of scienti c programming is evolving to larger, composite applications that are composed of smaller element applications. These composite applications are more frequently being targeted for distributed, heterogeneous networks of computers. They are most likely programmed by a group of developers. Software component technology and computational frameworks are being proposed and developed to meet the programming requirements of these new applications. Historically, programming systems have had a hard time being accepted by the scienti c programming community. In this paper, a programming model is outlined that attempts to organize the software component concepts and fundamental programming entities into programming abstractions that will be better understood by the application developers. The programming model is designed to support computational frameworks that manage many of the tedious programming details, but also that allow sucient programmer control to design an accurate, high-performance application.

Subject Terms

Document Classification unclassified	**Classification of SF298** unclassified

Classification of Abstract unclassified	**Limitation of Abstract** unlimited
Number of Pages 14	

A COMPONENT-BASED PROGRAMMING MODEL FOR COMPOSITE, DISTRIBUTED APPLICATIONS

THOMAS M. EIDSON*

Abstract. The nature of scientific programming is evolving to larger, composite applications that are composed of smaller element applications. These composite applications are more frequently being targeted for distributed, heterogeneous networks of computers. They are most likely programmed by a group of developers. Software component technology and computational frameworks are being proposed and developed to meet the programming requirements of these new applications. Historically, programming systems have had a hard time being accepted by the scientific programming community. In this paper, a programming model is outlined that attempts to organize the software component concepts and fundamental programming entities into programming abstractions that will be better understood by the application developers. The programming model is designed to support computational frameworks that manage many of the tedious programming details, but also that allow sufficient programmer control to design an accurate, high-performance application.

Key words. software components, computational frameworks, scientific applications, computational grids, distributed computing

Subject classification. Computer Science

1. Focus. Programming efficiency has been a problem in the scientific community for many years. Attempting to extract good execution performance from state-of-the-art high-performance architectures can be very time consuming. Distributed computing, especially on Grids [4], makes the situation worse as heterogeneous computing environments at multiple sites necessitate that a large amount of detail must be managed by the programmer. The situation is further complicated by the fact that larger, composite applications are becoming more common [7] [9].

Programming systems are needed to assist with managing this detail. The programming models will likely be based on higher-level abstractions than currently are common. In this note, a programming model is motivated that focuses on developing composite applications targeted for Grid environments. This is not an attempt to define a precise programming language as semantics are only defined to clarify the basic abstractions. Emphasis is on pragmatics, the relationship of the proposed abstractions and their meaning to the programmer [5].

The more relevant composite applications to the programming model discussed in this note are large, scientific applications that often include high performance computations. These applications typically couple smaller element applications that focus on a narrow aspect of the larger problem. These composite applications become challenging from the programming standpoint when the coupling of elements is tight and a significant amount of data and event transfers are required. Such coupling can be associated with both data-parallel and task-parallel application designs. The proposed model focuses on programming the task-parallel aspects while still allowing data-parallel code to be included.

*ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199 (email: teidson@icase.edu). This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the author was in residence at ICASE.

These proposed ideas also reflect a change in the usage of scientific applications. Until the last few years, most scientific applications were developed as a stand-alone package by an individual. Codes were shared infrequently as results were usually passed via reports. But, scientific programming is maturing and the free-lance programming styles are being replaced by best-practice styles. The resulting increased confidence in codes written by others and the need to build composite applications has led to increased code sharing.

2. Definitions.

1. A *programming model* is a set of abstractions and a set of rules that specify the combination of those abstractions in a form that can be translated to create execution instructions for an application.
2. A *Problem-Solving Environment* (PSE) is an integrated collection of software tools that facilitates problem-solving in some domain. This includes defining, building, executing, and managing the application. Additionally, this can include viewing and analyzing results related to the problem being solved.
3. A *computational framework* is an integrated collection of software tools that facilitates the development and execution of an application. A framework is the core feature of some PSEs.
4. An *element application* is a code in stand-alone executable or library form, that is focused on a relatively narrow aspect of some physics, mathematics, graphics, or other science.
5. A *task* refers to a set of user code with one or more entry points (functions, subroutines, methods, executables).
6. A *context* is defined as a collection of tasks and data packaged for execution and interaction with other tasks. A Unix process is an example of a context.
7. A *platform* is one or more computers managed as a single entity that is connected via a network to other platforms.
8. A *composite application* is a collection of tasks that would benefit from being distributed among several contexts located on several platforms. The application typically includes a range of data and event transfer operations between the various tasks and contexts that make-up the application. For this discussion, a data-parallel code is considered as one task that runs in one distributed context on one multi-node platform. A composite application is generally built by combining several element applications together under the control of a *work-flow* description.

   A composite application can be more than just a group of related element applications that share files. The element applications can be loosely or tightly coupled via a variety of data and event transfers. The nature of this coupling is a crude measure of the complexity of a composite application. Physical problems with non-linear, stiff behaviors often result in computer algorithms with complex communication patterns. The data and event transfers can also be viewed as *programming entities* that combine with the element applications to create a composite application. Non-traditional programming features such as computer resources, file systems, network performance, usage permissions, and user interfaces can also be viewed as programming entities.
9. *Metadata* is information about some programming entity that supports its use in some more comprehensive program (or *meta-program*) such as a composite application. Metadata includes interface specifications that describe how to access the programming entity and behavioral specifications that describe conceptual and practical details of correctly integrating the entity into the meta-program. For example, the information expressed in a Fortran subroutine could define some numerical algorithm. Interface metadata would describe the arguments needed to call that subroutine, typically in some general language. Behavioral metadata might describe the parallelization strategy as it relates

to target machines [3]. Behavioral metadata could even be used to describe physical and numerical assumptions embedded in the numerical algorithm.

10. A *software component* is a basic unit of software packaged for use in efficiently building some larger composite application. The software package includes metadata that minimally defines any interfaces to that software so that some computational framework can more easily provide the necessary integration. Software component technology is intended
    - to support software reuse and sharing,
    - to simplify use of multiple languages,
    - to support the efficient building of large applications, and
    - to assist building distributed applications.

    Eventually, this technology could enable plug-and-play environments for coupling reasonably complex physics. However, this will require research in a number of disciplines, well beyond the scope of this paper.
11. A computational *Grid* is a collection of heterogeneous computational hardware resources that are distributed (often over a wide area) and the software to use those resources. An important feature that converts a set of computers and software connected by an internet into a Grid is a set of support services (resource management, remote process management, communication libraries, security, monitoring support, etc.) and an organizational structure that provides usage guidelines or rules.

3. Software Components. One solution to providing the flexibility and efficiency needed to develop composite applications is software component technology. The basic technology has been demonstrated by several commercial products [8]. These products are not particularly suited to the scientific programming needs [1]. In general, the communication performance of these systems is poor to mediocre. They also do not support all the data types, programming languages, and computer systems that are common in the scientific community. Another problem is that they tend to include lots of baggage (e.g., business related services) that is not particularly useful to scientific applications. Scientific requirements tend to evolve constantly because of the importance of research and development. A solution that can grow on top of a light-weight core technology is needed. New ideas need to be rapidly integrated without disrupting ongoing work. Finally, the commercial model is based on application developers creating components and applications that can be packaged for use by others. This scenario exists to a lesser degree in the scientific community as researchers will need to constantly develop experimental components. A scientific computational environment will need more emphasis on efficient component and composite application development.

The Common Component Architecture (CCA) Forum [2] is developing a scientific software component specification along with several prototypes to test various approaches. This effort has made good progress at developing a system that meets the above requirements. The CCA Forum has defined a lightweight core design that is very promising. Each of the various prototypes experiment with implementations that focus on different aspects of target scientific applications. One aspect that is not well-addressed is a general programming model for composite applications. This paper attempts to define such a model.

The CCA specification is still evolving. The CCA focuses on defining three types of entities: Components, Ports, and Frameworks. The CCA Component is a set of data and code that provides some related functionality and that is chosen to provide a convenient programming granularity. A CCA Port is used to define all interfaces between different CCA Components. A CCA Framework is a software system that provides the functionality needed to couple CCA Components via CCA Ports to form a meta-program, all based on the CCA specification. A CCA Service is a CCA Component that implements some common func-

tionality that is generally associated with a CCA Framework for use by a user-created CCA Component. CCA Services include communication, discovery, and error handling among others. The CCA Port allows a CCA Component to define the specifications of an interface interaction without naming a specific instance of a CCA Component to which it will be connected. The connection is programmed separately.

4. A Programming Model.

4.1. Approach. Most composite, distributed applications can be implemented via a set of software components that are controlled by remote requests from some work-flow program or component and by using the services provided by some framework. The role of the framework is to manage the connections, which can be local and remote, between the various components and to provide functionality that facilitates the programming and execution of the application. A model based on this simple view provides all the features needed to build most applications but is not particularly satisfactory from a programmer's viewpoint.

The primary problem is that the above approach is too general. A good programming model needs to balance programming flexibility with programming efficiency [5]. Programmers like for the programming abstractions to suggest good programming construction while also providing flexible control for a range of options. Ideally, programming abstractions should match the concepts that a developer has used to design an application. Additionally, modern programming models need to support team programming. This means that programming intent needs to be expressed clearly and concisely.

A typical composite application will be composed of tasks (functions and methods), shared memory (internal task, global variables), files (external variables), and events (synchronization of behavior), which are organized into different contexts (processes) and executed on different platforms (computers). Each of these abstractions will need to be referenced in the user code as well as within the scope of the framework being used. As an application will contain one or more unique members of each of these abstraction types or families, it is clear that each family member needs a unique identity to create a precise application. Additionally, each family member may have multiple instances created as part of the executing application and each instance needs an identity.

Figure 4.1 shows an example application with two tasks. Task C runs in a "console" context (i.e., a process on the user's desktop) and Task R runs on a "remote server" context (i.e., a process on some remote computer). Pseudo-code for Task C that defines the work-flow is shown in Figure 4.2. In this example, Task C creates two contexts, X and Y, each of which have been configured to include an instance of R—R.1 and R.2. X and Y may possibly execute on two different remote computers. Task C then discovers a handle to each instance of Task R. A handle is just a referencing variable that the framework library functions create and use to insure that the correct instance of a task (or other similar family member) is used. Task C then repeatedly requests that R.1 and R.2 be executed concurrently inside a loop until the work-flow objective is satisfied. Task programming can also be multi-layered as either instance of Task R can also make framework requests. This example also shows a relatively simple strategy for programming multi-threaded distributed applications. Initiation of some "use" procedure for one or more instances of different Programming Components can be started concurrently. A "wait" procedure can be called at some later appropriate point before accessing the results of that "use". Each "use" can be managed by the framework via a separate thread.

In a distributed application, a server context is often used to run one or more tasks. A control loop (Figure 4.1) in the server context executes continually and waits on signals to start the execution of the code encapsulated by a task. This is often more efficient than starting each task in a new context.

The above task programming concept is similar to the component programming concept found in most

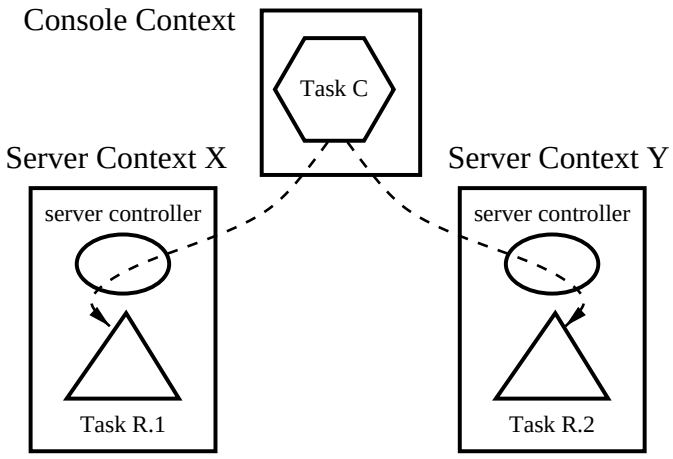

FIG. 4.1. *Task Programming in a Framework*

```
Arguments ri[2], ro[2]
Handle HX, HY, HR1, HR2

HX = Create_context("X")     // HX - a handle for Context X
HY = Create_context("Y")     // HY - a handle for Context Y
HR1 = Discover_task("R",HX)  // HR1 - a handle for Task R.1
HR2 = Discover_task("R",HY)  // HR2 - a handle for Task R.2

while(1) {
 Create_input(r1)
 Execute_task(HR1,ri[1])     // Execute_task - "use" procedure
 Execute_task(HR2,ri[2])

 <other computations>

 Wait_on_elements(HR1, HR2) // Wait_on_elements - "wait" procedure
 ro[1] = Get_results(HR1)
 ro[2] = Get_results(HR2)
 if (Results_satisfactory(ro)) break
}
```

FIG. 4.2. *Pseudo-code for Work-flow Program*

software component systems, both commercial and prototype. One problem with many software component systems is the tendency to hide any specifics about the context in which a specific software components executes. For well-developed applications where the user is merging loosely coupled components, this approach is satisfactory. But, science is about complex phenomena in which low-level constructs are tightly coupled and high-fidelity applications are generally needed to best model complex phenomena. If high-fidelity application development is to benefit from software component technology, programmers will need to have

some control over the location of tasks. Historically, the organization of data layout and the control of data transfers has always been the focus of performance optimization for scientific applications. It is difficult to believe that heterogeneous networks with widely varying performance levels and composite applications created from a smorgasbord of codes can be successfully used with a programming model where data flow can be ignored.

4.2. Description of a Programming Component. Based on the ideas expressed in the previous section, a programming model is proposed. The model primitives, referred to as Programming Components, will encapsulate user-defined code, data, and events along with desired programming services. A Programming Component will consist of an entity, both physical and abstract, along with metadata that can describe information needed to locate, create, and execute any software and hardware related to the application. Programming Components are selected to best match the concepts typically used by application developers to express their intent when designing a composite, distributed application. Programming Components will generally map to most software component designs. Herein, the CCA Model has been specifically targeted.

Programming Components enable the specification of all the entities of a component-based application in a compact and portable form. Traditional applications and the entities from which they are built are primarily defined by a set of code and files along with documentation on how to execute their interfaces. While these definitions can come in many forms, they all are reasonably compact mainly because they are programmed using a single programming language. Composite applications are more complex for no other reason than they are typically created from a greater number of element applications that are unfamiliar to the developer. When multiple languages, heterogeneous computer architectures and operating systems, and distributed computing environments are involved, a large amount of organizational detail needs to be included in the application definition. This information is fundamentally different from the algorithmic details that is the primary content of user code. The proposed Programming Component model is intended to provide a mechanism to express, separately via metadata, this organizational detail in a manner that a computation framework can best glue the different entities into a composite application. The metadata will be referred to as a Shared Programming Definition (SPD) to emphasize its value in defining an application outside the scope of a framework.

4.3. Extension to other Entities. In Figures 4.3, 4.4, and 4.5, an example is shown where information transfer is done separately from the control flow. The control flow would be similar to that shown in Figure 4.2. In this case, the second computations in Task S have some dependency on the first computations in Task R. Typically for performance reasons, it is sometimes desired to have the information flow be separate from the control flow. In other words, it is faster to let Task R notify Task S directly. This is often true for both data transfers and event signals.

When the two Tasks, R and S, are written, it is important to use a programming model and style that supports the development of an accurate and efficient application. There needs to be assurance that the framework providing the communications knows that Task S should be notified when Task R signals Event E. A Registry Service can be used for Task R to publish the existence of an instance of the event and for Task S to discover it, but this does not resolve the programming dilemma. The programmer of Task S needs to know the identity of Event E to program a discover request. Also, the use of Event E in Task S should be based on an understanding of the meaning of Event E; i.e., a behavioral specification of the event. While the above information could be shared by word of mouth, a more formal representation will be essential if associated with the development of a large composite application where tens or hundreds of programming entities are involved. Additionally, this information or event metadata needs to be associated with Task R.

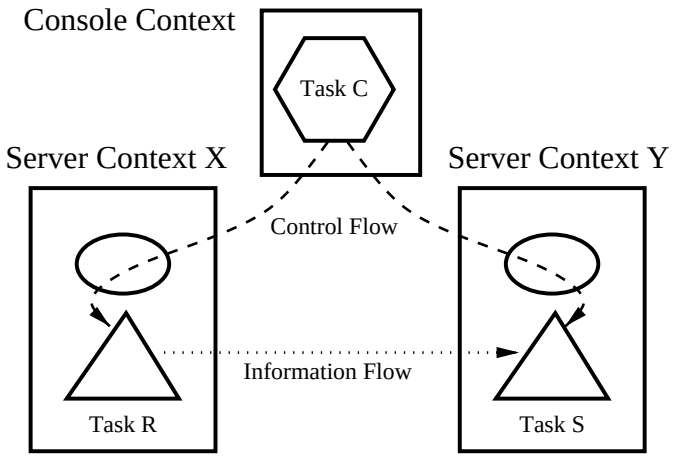

FIG. 4.3. *Information and Control Flows*

```
Handle HY, HR, HS, HE

HY = Discover_context("Y")
HR = Discover_this_task()
HS = Discover_task("S",HY)
HE = Publish_Event("E",HR)
 <first computations>
Signal_event(HE, HS)
 <second computations>
```

FIG. 4.4. *Pseudo-code for Task R*

```
Handle HX, HR, HE

HX = Discover_context("X")
HR = Discover_task("R",HX)
HE = Discover_Event("E",HR)
 <first computations>
Wait_on_event(HE)
 <second computations>
```

FIG. 4.5. *Pseudo-code for Task S*

If Task R is used in multiple applications, the Event E metadata needs to be available to the programming team for each application.

But, an event is not owned by a piece of code. Several Tasks could generate an event with the same meaning. From a code development prospective, the programming of an event is no different than the programming of a call to execute a function or task. Both provide functionality (behavior) of a specific program entity (with identity and state) that can be requested (via an interface) and that is outside the

scope of the current code. The coding of this request creates a dependence (relationship) between the current code or component and some external code, event or other programming entity. More precisely, a Programming Component is defined as an abstraction describing a well-defined programming entity that possesses the following properties.

- behavior
- identity
- an interface
- state
- relationship

Behavior is the value provided by a Programming Component. A Programming Component cannot be accurately used without a well-defined behavior. Particularly for scientific applications, behavior goes well beyond an understanding of how to use an interface. A programming model that emphasizes interface specifications trivializes the assumptions and concepts that were used to develop code being accessed via the interface. The primary purpose of the interface is to allow precise control for the behavior of another Programming Component to be accessed. State is important to allow for a wide range of functionality to be provided by a minimum number of Programming Components. State can refer to configuration where information in metadata can be altered to affect the behavior of an entity. State can also refer to internal variables within the entity that affect behavior during execution. These are important since they are typically controlled via an interface. However, state can also refer to the physical condition of some Programming Components. A piece of code could be in source or object form. A computer could be on or off line. A file could be in a readable form or encrypted. Additionally, Programming Components can have relationships. A piece of code that reads a file has a programming relationship with that file. The programming of the composite application is not complete until the correct instance of a conceptual file (or a procedure to determine the correct instance at runtime) is identified. Additionally, programming is needed to assist the framework in putting the correct instance in a location that is accessible by the code. Finally, the need for identity is the primary guideline used to determine if an entity or functionality will be a useful Programming Component.

Some software component systems tend to emphasize the independence of each component. Such a characteristic provides a very flexible programming environment. For practical reasons, a programming model should reflect the nature of its target applications. The physics' models simulated by scientific applications include coupled constructs and as such the most natural and useful component granularity will sometimes result in component dependencies.

4.4. Proposed Programming Components. Examples of several Programming Component families are given below. The Shared Programming Definition for each Programming Component family includes a unique identity along with configuration specifications. These specifications include interface, behavioral, and reference details. Each Programming Component would be programmed by calling some framework library in a manner suggested by the pseudo-code shown in the previous figures. The interface details would describe how to program the variable portion of the framework library functions. A common example is the argument list of a software component method execution. An Interface Definition Language (IDL) is often used to specify the interface details [6] [8]. The SPD approach would extend the IDL approach to allow the specification of key information relating to how a Programming Component accomplishes its results. Such behavioral specifications may include computational details (a data-parallel code), numerical details (the type of algorithm), or physical details (model equations include certain assumptions). Finally,

information specifying related Programming Components is needed to insure that all families of a coupled set of Components are included in an application. This could be another Task Component that computes required inputs with sufficient accuracy or that must be run concurrently to exchange information. Other references may restrict a Task Component to certain Platform Components that defines a set of acceptable computer characteristics for that Task.

The code needed to execute the functionality defined for each Programming Component will generally be part of Service Components supplied by the framework. The SPD of each Programming Component provides any configuration details for the Service Component. The Task Component is an exception as it also encapsulates user-written code.

- Task
  A Task Programming Component maps to the basic task or software component concept described above. It is the abstraction that encapsulates user-written code. A general model would allow the user code to be in source, object, or executable form. The user code could be single-threaded, multi-threaded, or data-parallel. The Computational Framework may or may not have control over any concurrent code, depending on the available framework services.
- Data Set
  Scientific applications deal with very large data sets. Often there is not sufficient memory to make a copy of a data set. Scientific applications also will use several independently written codes to operate on such a data set. These independent codes could be packaged in different Task Components. If one Task owns the data set, then programming flexibility will be a problem. A special Task Component could be used to own this data set. However, a detailed description of the data set would be needed in the SPD. Rather than overload the Task SPD, a separate Data Set Programming Component, that is focused on the data management role, is included in the model. Framework services would be available that allow data to be copied and linked into and out of the scope of Task and Data Set Components that are in the same context. Services would also provide data transfers between different contexts.
  When the data set sizes are small enough to allow several Data Set Component instances to exist in the same context, other programming flexibility benefits are available. One Data Set instance can be locked for use by an executing Task while another instance can be part of a data transfer. This allows the overall composite application to be programmed in a loosely coupled style as the code execution and data transfer may be triggered by work-flow in different locations.
- Event
  An event can be used to relay both control and data information. The encapsulation of the event concept as an Event Programming Component is done to provide a lightweight programming alternative to remote executions and data transfers. The Event Component behavior is intended to be implemented using an Event Service Component, which supports direct Task-to-Task signals as well as a message board to support event buffering. Broadcasts to groups should also be supported.
- Context
  Scientific codes often require performance solutions and distributed applications will generally benefit from optimized solutions. Organizing Task and Data Set Components for optimal locality to minimize data transfers may change in detail for different network scenarios, but it will almost always be important. A Context Programming Component can be used to group Task and Data Set Components so that they execute in the same context. A Context Component can be used similar to

an object in most object-oriented languages. It encapsulates methods and data together to support good overall program organization, but it allows codes written in different languages to be efficiently coupled in the same context.

- Platform

  One modern distributed computing goal is to relieve the user of the need to be concerned about the specific computer on which a code is run. But, performance concerns necessitate programmers specifying the locality of Tasks. Some locality concerns can be solved by the Context Component. However, the cost of transferring data between different Contexts can vary significantly and additional control is needed. Sometimes different Tasks may need to be in separate Contexts but still close together, where close is defined based on network performance. They may need to be on the same specific computer or just on two computers that are close. Other practical concerns also exist such as license issues, locality of file resources, architecture-specific coding, and proprietary restrictions.

  A Platform Programming Component is used to define a virtual computing resource on which a Context Component is to run. It can be defined by a specific IP address or it can refer to a pool of computers that is selected at runtime by some framework or operating system software based on requirements specified in a SPD. A Platform Component will tend to be hard-wired in the SPD of the Context Component. However, some applications will need to map a Context to a Platform at runtime based on parameters, such as grid size, that are passed to the related Context, Tasks, and Data Sets.

- Site

  Performance and organization have been a recurring theme in the above discussions. System software can only make good decisions when given sufficient information. Communication performance is highly dependent on the type of network over which it travels. Ultimately, network metadata will be needed for a framework to make optimal performance choices for an application. A rough estimate of network performance can be made by assuming that all computers on the same local area network (LAN) are "closer" together than those on different LANs. Additionally, information relating to file system organization can be useful to a framework. This includes defining file servers and cross-mounted file systems.

  A Site Programming Component is used to define a group of computers (or Platform Components) located on the same LAN. The Site SPD includes information about file systems, compilers and schedulers.

- File

  A File Programming Component is useful for many of the reasons that a Data Set Component is needed. A File Component just encapsulates data stored outside the scope of a Context Component. This can include the standard file concept, but also can be used to define information retrieved from a database or other entity. Similar to a Platform Component, a File can represent concrete entities or it can represent virtual information that must be chosen at runtime. Like a Data Set Component, it is useful to include specific format information in its SPD. This will allow Service Components to provide File to Data Set transfers via simple interfaces.

- Application

  An Application Programming Component is useful to package the complete definition of an application. It would include a list of all necessary user Programming Components and any framework

requirements. Work-flow, data-flow, and other organizational information as mentioned above is included. Alternative Components, usage suggestions, and documentation can also be included that will create a complete package.

The above items provide a reasonably complete set of primitives needed to fully define many composite applications. Other programming entities will also be useful but were not explicitly defined in this paper. These include user information to support access permission needs, separately started user interfaces that need to join an application, check-pointing support to define safe stopping points, and interactive support to aid in debugging, monitoring, and steering. The key feature that determines the need for a specific Programming Component is identity. Specifically, a Programming Component will need to be referenced, directly or indirectly, in multiple places in a user's code for the purpose of requesting desired functionality from a framework.

A side objective of Programming Components and their Shared Programming Definitions is to support portability from one framework to another. Ideally, one would like to move an application between frameworks with no configuration required. An appropriate choice of Programming Components should at least result in a minimal configuration needed by a framework in addition to importing the SPD information.

4.5. Framework Services. One of the positive features of the CCA approach is that the framework services can be built as CCA Components. This should allow a versatile set of services to evolve. These services are key to the success of the component approach. Services can encapsulate computer and computational science functionality that is needed by sophisticated applications, but which is viewed as burdensome overhead by the application developer. Application scientists just do not have the time to become skilled in all aspects of an application. A set of primary services needs to be designed that provide simple interfaces to a limited, but commonly used set of functionality. Advanced services can then be developed to replace the primary service when specialized functionality is needed.

Example of base framework services include the following.

- A registry service is needed to share the existence and location of Programming Component instances.
- Management services are needed to create, to destroy, and to otherwise manage remote processes and threads.
- Runtime services are needed to buffer information such as logs, queues, status, and availability.
- Interactive services are needed to provide monitoring, steering, and debugging functionality.
- Communication services are needed to provide configurable connections with appropriate performance and functionality.
- Information services are needed to store Shared Programming Definitions for convenient access by a target group.

Base framework services are only the starting point. Individual programming communities can build services that are specific to some numerical technique or physics. For example, a discipline data service could be created that defined several standard data formats and translation services between those formats. Task Components could be created to work with some or all of the formats. Two Tasks that used different formats could be integrated using a translation service.

4.6. Ideas Relating to Program Organization. Clearly, a good programming abstraction will collect lower-layer detail in a modular form that has beneficial organizational effects. Generally, models with minimal interactions between abstractions are easier to understand and to learn. However, it is those interactions that typically provide useful, sophisticated functionality. The choice of the content of a component

must be carefully chosen to find an appropriate balance between clarity, performance, and programming efficiency.

Most composite applications will have a main program or script that orchestrates the execution of the entities represented by the various Programming Components. This is a primary part of the application work-flow that defines all instructions to the framework. The work-flow is not restricted to a main program (say a Console Task Component) as it can be distributed throughout the user Task Components. The work-flow can also be described via application metadata. For example, the Application SPD could include a list of Context Components of a server type that should be started by the framework before the Console Component is started. The use of desired framework services can be specified in the work-flow.

While the application is basically defined by the work-flow, the complex nature of scientific applications will sometimes result in application designs with information or data flows that do not follow the control flow implemented by a primary work-flow program. A feature of some Software Component systems is the concept of a port. Each Task Component defines one or more ports to represent virtual interactions with external entities. For example, a function call would be made by referencing a specific port that represents an interface of some Programming Component, but not a particular instance. A separate step is needed to tie this "uses port" to an instance of a Programming Component that implements and registers a "provides port" of the same type that interfaces to the desired function. The port concept is not limited to function calls as it can be used to define all external interactions such as data transfers, event signals, and input and output with a file. It is useful to label the "function call" example as a work-flow port and the second group as data-flow ports. This labeling expresses a message of intent between the programmer and the system developer to infer that data-flow ports need higher performance. The coupling of the ports provides the last piece of the application definition. Static connections can be provided as part of the Application SPD, but a Registry Service Component is useful to provide dynamic connections.

While specific design choices will depend on the nature and goals of each application, most composite applications will be large and performance will be important. Frequently, the critical requirement in achieving good execution performance is the data flow. Control flow models where data or data variable references follow the execution path from function to function is one popular approach. However, a large amount of scientific codes also use separate data flow paths to transfer access to data between functions that are far apart in the calling sequence space. For example, common blocks in Fortran and global scope in C have provided this functionality. The labeling of ports as mentioned in the previous paragraph will support this mixed control/data flow style.

4.7. Compiler Issues. To access the behavior of a Programming Component, the programmer will need to include a call to some function or method. One approach is to provide a library that can be programmed by passing the identity of a Programming Component instance along with some parameters to request a specific desired behavior. This approach can be augmented with configuration details located in the metadata associated with the related Programming Component family. This dynamic style provides a great deal of flexibility. However, a number of Software Component systems use an Interface Definition Language (IDL) to store the metadata. An IDL compiler can then be used to generate a stub function that is loaded with the calling code. The user's code must also be modified to call this stub function. The stub function approach provides a custom interface to the Programming Component and it allows for potentially better performance as the overhead of using component methodology can be reduced to one extra function call if the calling and the requested code are in the same context. Because of the tendency of composite applications to be distributed and the desire for execution flexibility, it is suggested that a dynamic, library

style should be the target model. Where performance is critical, the stub function approach can be provided as an optimization.

The programming flexibility suggested in the previous paragraph leads to another suggestion. Current Software Component systems tend to treat traditional compilation as distinct operation in the building of a component or composite application. Clearly, there is much to be gained from compilers that are built around component concepts. At a minimum the need for a separate IDL compiler would disappear. All Programming Component behavior requests could be programmed via framework library calls, but using syntax recognized by the compiler. The compiler could then use the metadata associated with the Programming Component and any other Components with a relationship to generate the best code. Additionally, the metadata could contain compilation suggestions and execution history that could benefit the compilations of a Task Component, both directly and indirectly.

5. Summary. The general scientific programming community will require efficient software systems to successfully develop composite applications in Grid environments. The proposed ideas on programming models are intended to help bridge the gap between the software component ideas being developed and the practical nature of the computational scientist. The exact nature of the programming model and languages that need to be created will probably be the result of feedback between the system software developers and the computational scientist. However, prototypes with good programming models need to be made available to seed this feedback.

Acknowledgments. The author would like to thank the many scientific programmers and computer scientists who have discussed programming with him over the years. Also, the many contributions to the CCA Forum are appreciated as the CCA provided a foundation on which the author's ideas could be expressed.

REFERENCES

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolenski, *Towards a common component architecture for high-performance scientific computing*, in Eighth IEEE International Symposium on High Performance Distributed Computing, August 1999.

[2] CCA, *Common Component Architecture Forum webpage*, in http://www.cca-forum.org, 2001.

[3] C. Cicalese and S. Rotenstreich, *Behavioral specification of distributed software*, Computer, (1999), p. 46.

[4] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1995.

[5] J. Sammet, *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969.

[6] J. Siegel, *CORBA: Fundamentals and Programming*, John Wiley and Sons, 1996.

[7] J. Stewart and H. Edwards, *The SIERRA framework for developing advanced parallel mechanics applications*, in Proceedings of First Sandia Workshop on Large-Scale PDE-Constrained Optimization, Springer Lecture Notes in Computational Science and Engineering, 2001.

[8] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[9] R. Weston, J. Townsend, T. Eidson, and R. Gates, *A distributed computing environment for multidisciplinary design*, in 5th AIAA/NASA/USAF/ISSMO Symposium on Multiple Disciplinary Analysis and Optimization, September 1994.